

Lecture 8: O/O Programming

Topics

- OO Programming Languages
- Developing programs from Designs
 - Class and method definition
 - Structuring the set of program classes
- Testing for Object Oriented programs

OO Programming Languages

- Simula (Simulation)
- SmallTalk (AI Community)
- C++ (C)
- Eiffel (Software Engineering, CASE)
- Java (Devices)
- C# (Copyright Problems)

Sample Differences

<i>Concept</i>	<i>Smalltalk</i>	<i>C++</i>	<i>Java</i>
Garbage Collect.	√	×	√
Pointers	×	√	×
Inheritance	single	multiple	single
Interface Class	×	×	√
Root Object Class	√	×	√
Type checking	×(run)	√	√
Abstract Class	√(× enf)	√	√

Seminal O/O Ideas

- Information hiding
 - Construct modules/components where you access information via operations that hide how it was implemented
- Abstract Data Types
 - Original information hiding idea

Abstract Data Types

- Abstract objects and a set of operations on them
- Implementation is not part of definition
- Definition of object given by axioms involving objects and operations

List ADT Definition

	Constructors: $s =$	
observers	Nil	prepend(n, s')
null?(s)	True	False
head(s)	Error	n
tail(s)	Error	s'

ADT's and OO Classes

- Alternative points of view
 1. Classes define implementations of ADT specifications
 2. ADT's are for simple algebraic objects such as numbers and lists. Classes are for more complex structures.
 3. Alternative forms of abstraction
 - ADT's: type abstraction
 - O/O Classes: procedural

Developing Programs from Designs

- Order of implementation strategy
- GUI layout details
- Class Definitions
- Method definitions

Order of Implementation

- Bottom up
- Top Down
- Least-Coupled First

Bottom Up

- Functional Design Strategy
 - Start with functions at the bottom of the design tree
 - Need *drivers* to carry out tests
- Application to OO Designs?
 - Objects which receive messages but do not send them
 - E.g. DS MemberData

Top Down

- Functional Design Strategy
 - Start with functions at the top of the design tree
 - Need *stubs* to carry out tests
- Application to O/O designs?
 - Objects which send messages but do not receive them
 - E.g. DS Start class, GUI Frame, etc.

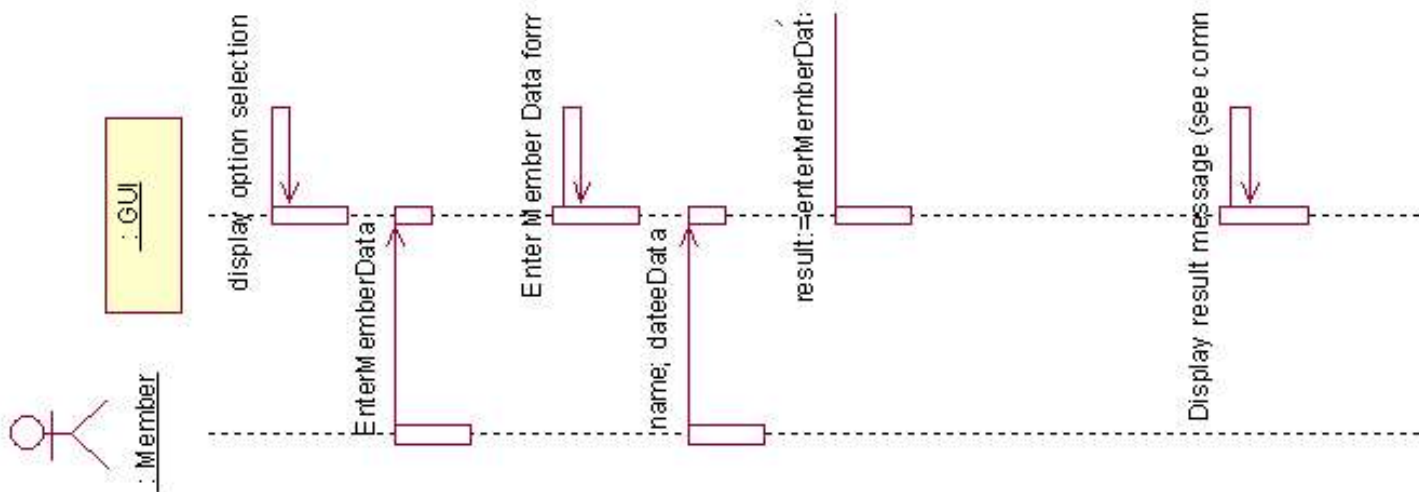
Least-Coupled First

- Rationale?
 - Easier to do unit testing
- E.g. DS
 - LogOn class in DomainLogic
- Our approach for DS: implement each subsystem separately: e.g. GUI, DL, DB

GUI Layout

- Information sources
 - Interaction sequence diagrams
 - Forms through which the user interacts
 - Collaboration diagrams
 - Frame and Dialog Objects used in Design

Screens and Interaction Diagrams



Enter member data scenario

Sample DS GUI Layout

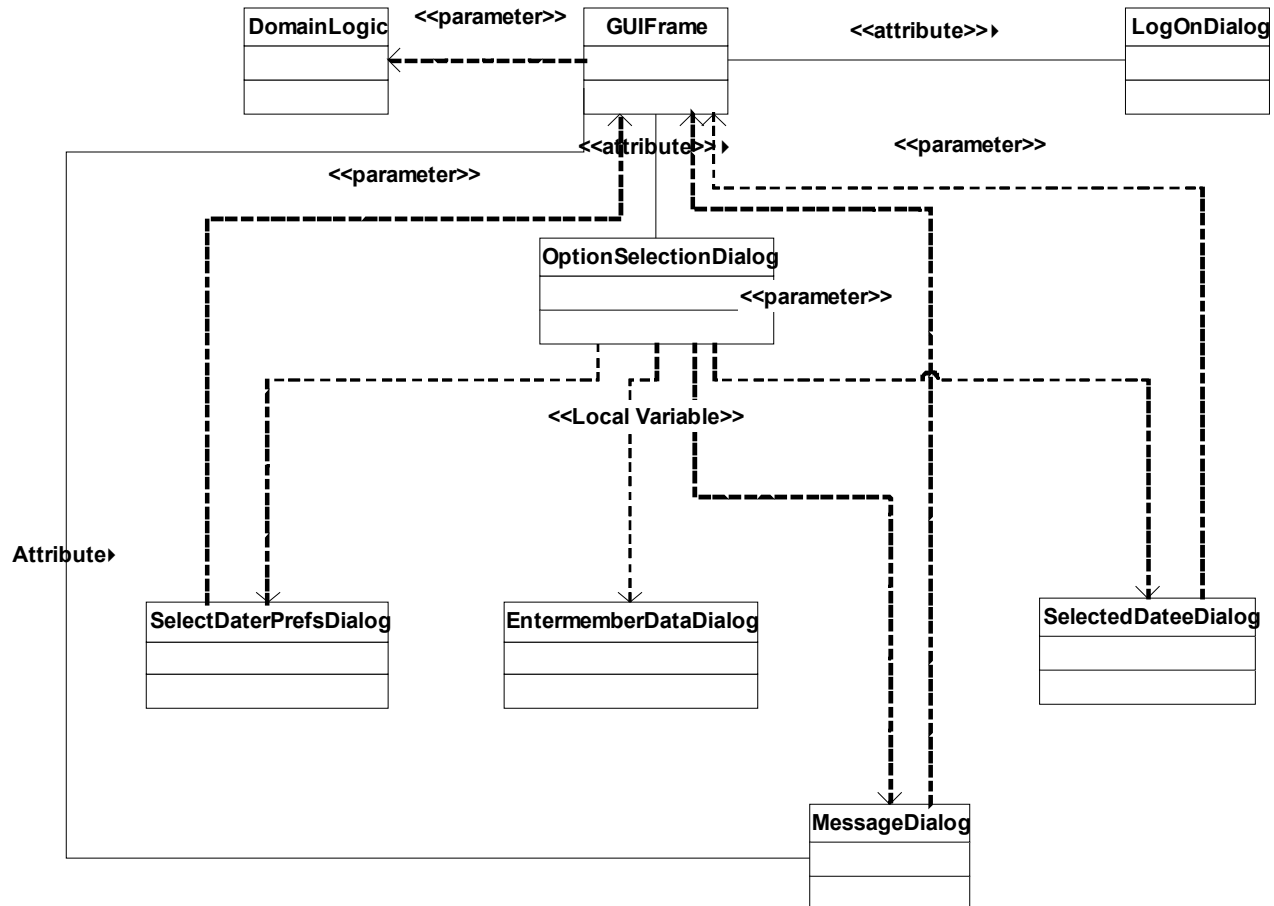


Class Definition

- Map from UML Class description to PL (Java) class definition
- Determine class visibility in class definitions
- Additional Details. E.g.
 - For GUI classes determine event management strategy
 - Additional classes

Dating system example

GUI class definition



LogOnDialog
+OKButton: Button +nameField: TextField -parentFrame: Frame
+show() +setVisible() +LogOnDialog(): LogOnDialog() +buttonAction(in id:buttonID)

DaterOptionSelectionDialog
+getADate: Button +setMemberData: Button -parentFrame: Frame
+setVisible() +show() +buttonAction(in id : int) +DaterOptionSelectionDialog(in gUIGuiFrame)

Class Visibility

- All dialog classes except Message will be defined as inner classes of GUIFrame
- Message will be top level globally visible class
- DomainLogic package will be imported, to get visibility to this class

GUI Event Strategy

- Need actionPerformed(ActionEvent e) methods for processing button pushes
- Declare that GUI (and the dialogs) implement ActionListener. Adopting strategy where we attach the event processing methods directly to the units that contain the event causing components

Additional Detailed Classes?

- E.g. Primitive data type parameters in Java are call by value
- Will need to define simple classes to allow call by reference

```
class BooleanRef
{
    public boolean val;
    public BooleanRef() {}
}
```

```

public class GUI extends Frame implements ActionListener
{
    private Button start;
    private Button end;
    public DomainLogic domainLogic;
    private LogOnDialog logOnDialog;
    private MemberCommandsDialog memberCommandsDialog;
    private AdminCommandsDialog adminCommandsDialog;
    private int userType;
    public String userName;
    private MessageDialog message;
    public GUI(DomainLogic dL) { ....}
    public void actionPerformed(ActionEvent e){ ... }

    class LogOnDialog extends Dialog implements ActionListener { ...}
    class MemberCommandsDialog extends Dialog implements ActionListener { ... }
    class AdminCommandsDialog extends Dialog implements ActionListener { ... }
    class GetMemberNameDialog extends Dialog implements ActionListener { ... }
    class SelectDaterPreferencesDialog extends Dialog implements ItemListener,ActionListener { ...}
    class EnterMemberDataDialog extends Dialog implements ActionListener { ... }
    class SelectedDateeDialog extends Dialog implements ActionListener { ... }
}

```

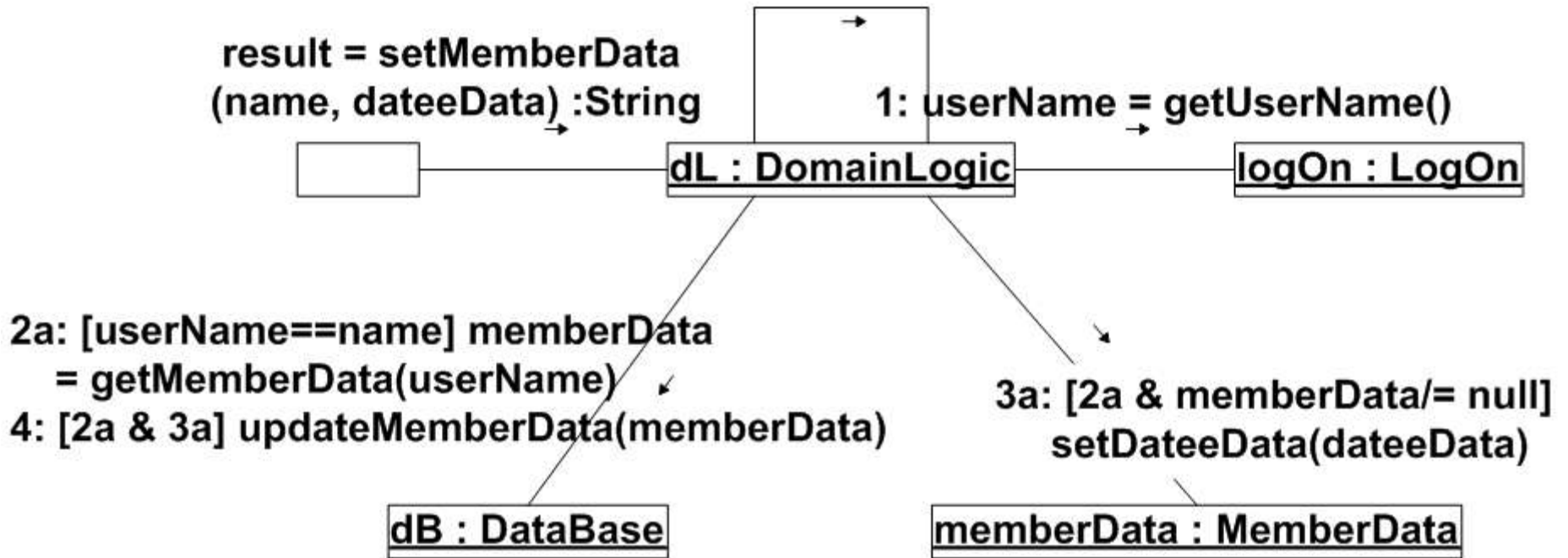
Method Definition

- Object receives a message -> corresponding class method
- Method logic:
 - Examine Collaboration diagram for object responses to messages

Dating system example

Domain Logic method definition

2b: [userName≠/ name] result := "Not logged on"
 3b: [2a & memberData == null] result = "NotAMember"
 5: [2a & 3a] result = "UpdateMade"



Domain (Business) Logic subsystem updates a member's data

```
String setMemberData(String name, dateeData dD)
{
    String userName;
    MemberData memberData;
    userName = logOn.getUserName;
    if (userName == name)
    {
        memberData = dB.getMemberData(userName);
        if (memberData != null)
        {
            memberData = memberData.setDateeData(dD);
            dB.updateMemberData(memberData)
            return "UpdateMade";
        }
        return "NotAMember";
    }
    return "notLoggedIn";
}
```

Introduction to Object Oriented Testing

- *Unit testing* Test individual classes or small groups of classes during development
- *Test first programming* Write the class tests before the class
- *Functional testing* Use test scripts that describe a sequence of program interactions and behavior verifications for system testing

Self Testing Classes

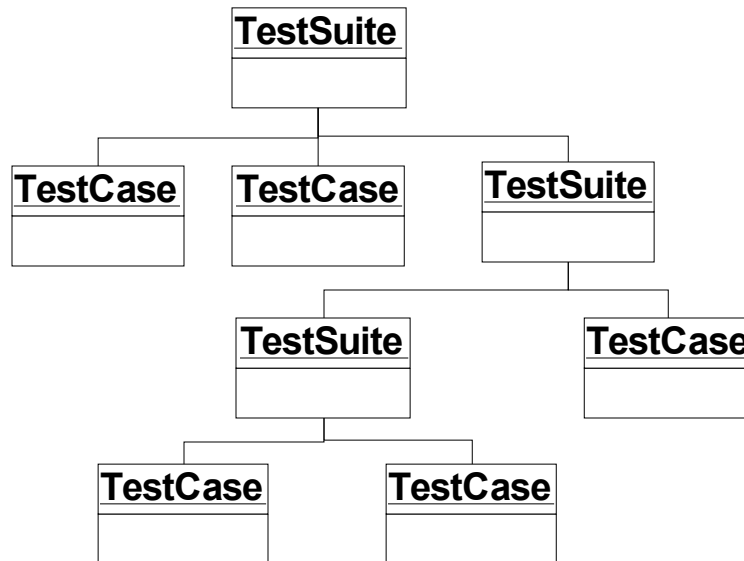
- Class has a main method that is used for testing
- Main method:
 - Creates an instance of the class
 - Calls class methods that change the state
 - Calls class methods that verify results of a test
 - May require special test methods in CUT (Class Under Test)

Test Case

- Collection of test and verify methods
- Main method
 - Creates instance of class under test
 - Performs initialization
 - Runs test methods
- Could be self-testing class or separate class

Tests and Test Suites

- Test = TestCase or TestSuite
- TestSuite = Collection of Tests



TestRunners

- Special class whose instances run tests
- Execute a TestRunner with Test Object
- Test runners capabilities
 - Fully automatic: execute all test cases
 - StepAndWait: user approval to go to next test case
 - Error reports: accept from test cases

J-Unit OO Unit Test Framework

- A TestRunner with a GUI interface
 - Executes a test suite
- Classes that you can use to set up test suites of test cases, to be run by TestRunner.
- E.g. TestSuite contains an add method for adding test objects to a test suite object
- E.g. test cases are instances of a class that subclasses TestCase and inherits a runTest() method that is called by TestRunner

J-Unit Usefulness

- Nifty use of OO mechanisms, in order to make it work
- Provides user with a nice interface for running tests
- Easier to create your own test suites and test cases, than to figure out how to use it